On the Behaviours Produced by Instruction Sequences under Execution

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 904, 1098 XH Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. The behaviour produced by an instruction sequence under execution is a behaviour to be controlled by some execution environment: each step performed actuates the processing of an instruction by the execution environment and a reply returned at completion of the processing determines how the behaviour proceeds. The increasingly occurring case where the processing takes place remotely involves the generation of a stream of instructions to be processed and a remote execution unit that handles the processing of this stream of instructions. We use process algebra to describe the behaviours produced by instruction sequences under execution and to describe two protocols implementing these behaviours in the case of remote processing. We also show that all finite-state behaviours considered in process algebra can be produced by instruction sequences under execution.

Keywords: instruction sequence, thread extraction, process extraction, instruction stream processing, instruction sequence producible process.

1998 ACM Computing Classification: D.1.4, D.2.1, F.1.1, F.1.2, F.3.2.

1 Introduction

The behaviour produced by an instruction sequence under execution is a behaviour to be controlled by some execution environment. It proceeds by performing steps in a sequential fashion. Each step performed actuates the processing of an instruction by the execution environment. A reply returned by the execution environment at completion of the processing of the instruction determines how the behaviour proceeds. Increasingly, the processing of instructions takes place remotely. This means that, on execution of an instruction sequence, a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. The main objective of the current paper is to bring this phenomenon better into the picture. To achieve this objective, we describe two protocols for instruction stream processing that deal with this phenomenon.

The phenomenon sketched above is found if it is impracticable to load the instruction sequence to be executed as a whole. For instance, the storage capacity of the execution unit is too small or the execution unit is too far away.

The phenomenon requires special attention because the transmission time of the messages involved in remote processing makes it hard to keep the execution unit busy without intermission. The more complex protocol for instruction stream processing described in this paper is directed towards keeping the execution unit busy. There is no reason to use the word "remote" in a narrow sense. It is convenient to consider processing remote if it involves message passing with transmission times that are not negligible. In that case, the more complex protocol provides a starting-point for studies of basic techniques aimed at increasing processor performance, such as pre-fetching and branch-prediction, at a more abstract level than usual. Therefore, we consider the more complex protocol relevant to the area of computer architectures.

The descriptions of the protocols for instruction stream processing start from the behaviours produced by instruction sequences under execution. By that we abstract from the instruction sequences which produce these behaviours. At the level of abstraction concerned, it is easy to describe how the instruction streams are generated. How instruction streams can be generated efficiently from instruction sequences is another matter.

The work presented in this paper is a spin-off of a line of research whose working hypothesis is that instruction sequence is a central notion of computer science. In this line of research, issues concerning the following subjects from the theory of computation have been investigated from the viewpoint that a program is an instruction sequence: semantics of programming languages, expressiveness of programming languages, computability, computational complexity, and performance related matters of programs. In the area of computer architectures, basic techniques aimed at increasing processor performance have been studied in this line of research as well, The main references to the work carried out in this line of research can for instance be found in [11].

In the work carried out in the line of research mentioned above, use is made of a special-purpose process algebra to describe and analyse the behaviours produced by instruction sequences under execution. The process algebra in question is introduced in [5] under the name BPPA (Basic Polarized Process Algebra), but prompted by the development of thread algebra [6], which is a design on top of it, BPPA has been renamed to BTA (Basic Thread Algebra). The behaviours considered in BTA are called threads. They model the behaviours produced by instruction sequences under execution directly: upon each action performed by a thread, a reply from an execution environment – which takes the action as an instruction to be processed – determines how the thread proceeds. In general-purpose process algebras, such as ACP [4,2], CCS [15,17] and CSP [12,16], it is rather awkward to describe and analyse behaviours of this kind. However, the behaviours considered in BTA can be viewed as processes that are definable over ACP. This allows for the protocols for instruction stream processing to be described using ACP or rather ACP^{τ}, an extension of ACP which supports abstraction from internal actions.

Process algebras such as ACP, CCS and CSP are considered relevant to computer science, as is witnesses by the extent of the work on process algebras

in theoretical computer science. This means that there must be programmed systems whose behaviours are taken for processes as considered in such a process algebra. Another objective of the current paper is to establish results concerning the processes as considered in ACP that can be produced by programs under execution, starting from the perception of a program as an instruction sequence. The main result established is that, by apposite choice of basic instructions, all finite-state processes can be produced by instruction sequences provided that the cluster fair abstraction rule (see e.g. [13], Section 5.6) is valid.

In the work carried out in the line of research mentioned above, use is made of an algebra of instruction sequences to describe and analyse instruction sequences. The algebra in question is introduced in [5] under the name PGA (ProGram Algebra). The instruction sequences considered in PGA are single-pass instruction sequences, i.e. a finite or infinite sequences of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. PGA does not provide a notation for programs that is intended for actual programming: programs written in an assembly language are finite instruction sequences for which single-pass execution is usually not possible. We show that all finite-state processes can as well be produced by programs written in a program notation which is close to existing assembly languages.

Instruction sequences under execution, and more generally threads, may make use of services such as counters, stacks and Turing tapes. The use operators introduced in [6] are concerned with the effect of services on threads. An interesting aspect of making use of services is that instruction sequences under execution that make use of services may produce infinite-state processes. On that account, we add the use operators to BTA and make precise what processes are produced by instruction sequences under execution that make use of services.

This paper is organized as follows. First, we give brief summaries of PGA and BTA (Sections 2 and 3) and define the threads produced by the instruction sequences considered in PGA (Section 4). Next, we introduce a kind of instructions/actions meant for interaction with services (Section 5), give brief a summary of ACP^{τ} (Section 6), and define the processes produced by the threads considered in BTA if the actions are of the kind meant for thread-service interaction (Section 7). Then, we describe two protocols for instruction stream processing (Sections 8 and 9) and discuss some conceivable adaptations of the more complex one (Section 10). After that, we introduce a kind of instructions/actions meant for producing non-deterministic processes (Section 11) and show how they allow for all finite-state processes to be produced by instruction sequences (Section 12). Following this, we extend BTA with operators for making use of services and define the processes produced by threads making use of services (Section 13). We also introduce PGLD, a program notation close to existing assembly languages, and show that all finite-state processes can also be produced by PGLD programs (Section 14). Finally, we make some concluding remarks (Section 15).

This paper consolidates material from [7,9,10].

2 Program Algebra

In this section, we review PGA (ProGram Algebra). The starting-point of program algebra is the perception of a program as a single-pass instruction sequence. The concepts underlying the primitives of program algebra are common in programming, but the particular form of the primitives is not common. The predominant concern in the design of program algebra has been to achieve simple syntax and semantics, while maintaining the expressive power of arbitrary finite control.

In PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces a reply at its completion. The possible replies are the Boolean values T and F.

PGA has the following *primitive instructions*:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* $!$.

We write $\mathfrak{I}$ for the set of all primitive instructions of PGA. On execution of an instruction sequence, these primitive instructions have the following effects:

- the effect of a positive test instruction $+a$ is that basic instruction a is executed and execution proceeds with the next primitive instruction if T is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one — if there is no primitive instructions to proceed with, deadlock occurs;
- the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed;
- the effect of a plain basic instruction a is the same as the effect of $+a$, but execution always proceeds as if T is produced;
- the effect of a forward jump instruction $\#l$ is that execution proceeds with the l-th next instruction of the program concerned — if l equals 0 or there is no primitive instructions to proceed with, deadlock occurs;
- the effect of the termination instruction $!$ is that execution terminates.

PGA has the following constants and operators:

- for each $u \in \mathfrak{I}$, an *instruction* constant u;
- the binary *concatenation* operator $\_ \,;\, \_$;
- the unary *repetition* operator $\_^{\omega}$.

We assume that there is a countably infinite set of variables which includes x, y, z. Terms are built as usual. We use infix notation for concatenation and postfix notation for repetition.

Table 1. Axioms of PGA

$(x\,;y)\,;z = x\,;(y\,;z)$	PGA1
$(x^n)^\omega = x^\omega$	PGA2
$x^\omega\,;y = x^\omega$	PGA3
$(x\,;y)^\omega = x\,;(y\,;x)^\omega$	PGA4

A closed PGA term is considered to denote a non-empty, finite or eventually periodic infinite sequence of primitive instructions.[1] Closed PGA terms are considered equal if they represent the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 1. In this table, n stands for an arbitrary positive natural number. The term x^n is defined by induction on n as follows: $x^1 = x$ and $x^{n+1} = x\,;x^n$. The *unfolding* equation $x^\omega = x\,;x^\omega$ is derivable. Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form t or $t\,;t'^\omega$, where t and t' are closed PGA terms in which the repetition operator does not occur.

The initial models of PGA are considered its standard models. Henceforth, we restrict ourselves to the initial model $\mathcal{I}_{\mathrm{PGA}}$ of PGA in which:

- the domain is the set of all finite and eventually periodic infinite sequences over the set $\mathfrak{I}$ of primitive instructions;
- the operation associated with $;$ is concatenation;
- the operation associated with $^\omega$ is the operation $^{\underline{\omega}}$ defined as follows:
  - if F is a finite sequence over $\mathfrak{I}$, then $F^{\underline{\omega}}$ is the unique eventually periodic infinite sequence F' such that F concatenated n times with itself is a proper prefix of F' for each $n \in \mathbb{N}$;
  - if F is an eventually periodic infinite sequence over $\mathfrak{I}$, then $F^{\underline{\omega}}$ is F.

The members of the domain of $\mathcal{I}_{\mathrm{PGA}}$ are called *instruction sequences*.

$\mathcal{I}_{\mathrm{PGA}}$ is loosely called *the* initial model of PGA because all initial models of PGA are isomorphic.

3 Thread Algebra

In this section, we review BTA (Basic Thread Algebra). BTA is a process algebra tailored to the description and analysis of the behaviours produced by instruction sequences under execution. These behaviours are called threads.

In BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic actions*, with $\mathsf{tau} \notin \mathcal{A}$, has been given. Besides, tau is a special basic action. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. A thread performs basic actions in a sequential fashion. Upon each basic action performed, a reply from an execution environment determines how it proceeds. The possible replies are the Boolean values T and F. Performing tau, which is considered performing an internal action, always leads to the reply T.

[1] An eventually periodic infinite sequence is an infinite sequence with only finitely many distinct suffixes.

Table 2. Axiom of BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = x \trianglelefteq \mathsf{tau} \trianglerighteq x$	T1

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with an additional sort in Section 13.

BTA has one sort: the sort $\mathbf{T}$ of *threads*. To build terms of sort $\mathbf{T}$, it has the following constants and operators:

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A}_{\mathsf{tau}}$, the binary *postconditional composition* operator $\trianglelefteq a \trianglerighteq : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

We assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z. Terms of sort $\mathbf{T}$ are built as usual. We use infix notation for the postconditional composition operators. We introduce *basic action prefixing* as an abbreviation: $a \circ t$, where $a \in \mathcal{A}_{\mathsf{tau}}$ and t is a term of sort $\mathbf{T}$, abbreviates $t \trianglelefteq a \trianglerighteq t$.

The thread denoted by a closed term of the form $t \trianglelefteq a \trianglerighteq t'$ will first perform a, and then proceed as the thread denoted by t if the reply from the execution environment is T and proceed as the thread denoted by t' if the reply from the execution environment is F. The threads denoted by D and S will become inactive and terminate, respectively.

BTA has only one axiom. This axiom is given in Table 2. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

Notice that each closed BTA term denotes a thread that will become inactive or terminate after it has performed finitely many actions. Infinite threads can be described by guarded recursion.

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort $\mathbf{T}$ and each t_X is a BTA term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur in E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [3].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ of sort $\mathbf{T}$ standing for the unique solution of E for X. We write $\langle t_X|E\rangle$ for t_X with, for all $Y \in \mathrm{V}(E)$, all occurrences of Y in t_X replaced by $\langle Y|E\rangle$. The axioms for the constants standing for the unique solution of guarded recursive specifications are RDP (Recursive Definition Principle) and RSP (Recursive Specification Principle), which are given in Table 3. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary BTA term, and E stands for an arbitrary guarded recursive specification over BTA. Side conditions are added to restrict what X, t_X and E stand for.

Closed terms of BTA extended with constants for solutions of guarded recursive specifications that denote the same infinite thread cannot always be proved

Table 3. RDP, RSP and AIP

$\langle X\|E\rangle = \langle t_X\|E\rangle$	if $X = t_X \in E$	RDP	$\pi_0(x) = \mathsf{D}$	P0
$E \Rightarrow X = \langle X\|E\rangle$	if $X \in \mathrm{V}(E)$	RSP	$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
			$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$		AIP	$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

equal by means of the axiom of BTA, RDP and RSP. We introduce AIP (Approximation Induction Principle) to remedy this. AIP is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a thread is obtained by cutting it off after it has performed n actions. AIP is also given in Table 3. Here, approximation up to depth n is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. The axioms for these operators are axioms P0–P3 in Table 3. P1–P3 are actually axiom schemas in which a stands for arbitrary basic action and n stands for an arbitrary natural number.

We write BTA+REC for BTA extended with the constants for solutions of guarded recursive specifications, the projection operators and the axioms RDP, RSP, AIP and P0–P3.

The minimal models of BTA+REC are considered its standard models. Recall that a model is minimal if the elements of the domains associated with the sorts can be denoted by closed terms. Henceforth, we restrict ourselves to the minimal models of BTA+REC. We assume that a minimal model $\mathcal{M}_{\mathrm{BTA+REC}}$ of BTA+REC has been given.

We use the term *thread* for the elements from the domain of $\mathcal{M}_{\mathrm{BTA+REC}}$, and we denote the interpretations of constants and operators in $\mathcal{M}_{\mathrm{BTA+REC}}$ by the constants and operators themselves.

Let T be a thread. Then the set of *states* or *residual threads* of T, written $Res(T)$, is inductively defined as follows:

- $T \in Res(T)$;
- if $T' \trianglelefteq a \trianglerighteq T'' \in Res(T)$, then $T' \in Res(T)$ and $T'' \in Res(T)$.

Let T be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then T is *regular over* $\mathcal{A}'$ if the following conditions are satisfied:

- $Res(T)$ is finite;
- for all $T', T'' \in Res(T)$ and $a \in \mathcal{A}_{\mathsf{tau}}$, $T' \trianglelefteq a \trianglerighteq T'' \in Res(T)$ implies $a \in \mathcal{A}'$.

We say that T is *regular* if T is regular over $\mathcal{A}_{\mathsf{tau}}$.

We will make use of the fact that being a regular thread coincides with being the solution of a finite guarded recursive specification in which the right-hand sides of the recursion equations are of a restricted form.

A *linear recursive specification* over BTA is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over BTA, where each t_X is a term of the form D, S or $Y \trianglelefteq a \trianglerighteq Z$ with $Y, Z \in V$.

Table 4. Defining equations for thread extraction operation

$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert \#l\rvert = \mathsf{D}$
$\lvert a\,;F\rvert = a \circ \lvert F\rvert$	$\lvert \#0\,;F\rvert = \mathsf{D}$
$\lvert +a\rvert = a \circ \mathsf{D}$	$\lvert \#1\,;F\rvert = \lvert F\rvert$
$\lvert +a\,;F\rvert = \lvert F\rvert \trianglelefteq a \trianglerighteq \lvert \#2\,;F\rvert$	$\lvert \#l+2\,;u\rvert = \mathsf{D}$
$\lvert -a\rvert = a \circ \mathsf{D}$	$\lvert \#l+2\,;u\,;F\rvert = \lvert \#l+1\,;F\rvert$
$\lvert -a\,;F\rvert = \lvert \#2\,;F\rvert \trianglelefteq a \trianglerighteq \lvert F\rvert$	$\lvert !\rvert = \mathsf{S}$
	$\lvert !\,;F\rvert = \mathsf{S}$

Proposition 1. *Let T be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then T is regular over $\mathcal{A}'$ iff there exists a finite linear recursive specification E over* BTA *in which only basic actions from $\mathcal{A}'$ occur such that T is the solution of E for some $X \in \mathrm{V}(E)$.*

Proof. This proposition generalizes Theorem 1 from [18] from the projective limit model of BTA to an arbitrary minimal model of BTA. However, the proof of that theorem is applicable to any minimal model of BTA. □

4 Thread Extraction

In this short section, we use BTA+REC to make mathematically precise which threads are produced by instruction sequences under execution.

For that purpose, $\mathcal{A}$ is taken such that $\mathcal{A} \supseteq \mathfrak{A}$ is satisfied.

The *thread extraction* operation $\lvert\_\rvert$ assigns a thread to each instruction sequence. The thread extraction operation is defined by the equations given in Table 4 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$, and $u \in \mathfrak{I}$) and the rule that $\lvert \#l\,;F\rvert = \mathsf{D}$ if $\#l$ is the beginning of an infinite jump chain. This rule is formalized in e.g. [8].

Let F be an instruction sequence and T be a thread. Then we say that F *produces* T if $\lvert F\rvert = T$.

5 Program-Service Interaction Instructions

Recall that, in PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. In the sequel, we will make use a version of PGA in which the following additional assumptions relating to $\mathfrak{A}$ are made:

- a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given;
- a fixed but arbitrary set $\mathcal{M}$ of *methods* has been given;
- $\mathfrak{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

Each focus plays the role of a name of some service provided by an execution environment that can be requested to process a command. Each method plays the role of a command proper. Executing a basic instruction of the form $f.m$ is taken as making a request to the service named f to process command m.

A basic instruction of the form $f.m$ is called a *program-service interaction instruction*. Recall that, in BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of basic actions has been given. In the sequel, we will make use of a version of BTA in which $\mathcal{A} = \mathfrak{A}$. A basic action of the form $f.m$ is called a *thread-service interaction action*.

The intuition concerning program-service interaction instructions given above will be made fully precise in Section 7, using the general-purpose process algebra ACP.

6 Process Algebra

In this section, we review ACP^τ (Algebra of Communicating Processes with abstraction). This process algebra will among other things be used to make precise what processes are produced by the threads denoted by closed terms of BTA+REC. For a comprehensive overview of ACP^τ, the reader is referred to [2,13].

In ACP^τ, it is assumed that a fixed but arbitrary set A of *atomic actions*, with $\tau, \delta \notin \mathsf{A}$, and a fixed but arbitrary commutative and associative function $|\,:\mathsf{A}\cup\{\tau\}\times\mathsf{A}\cup\{\tau\}\to\mathsf{A}\cup\{\delta\}$, with $\tau\,|\,e = \delta$ for all $e \in \mathsf{A}\cup\{\tau\}$, have been given. The function $|$ is regarded to give the result of synchronously performing any two atomic actions for which this is possible, and to give δ otherwise. In ACP^τ, τ is a special atomic action, called the silent step. The act of performing the silent step is considered unobservable. Because it would otherwise be observable, the silent step is considered an atomic action that cannot be performed synchronously with other atomic actions. We write A_τ for $\mathsf{A}\cup\{\tau\}$.

ACP^τ has the following constants and operators:

- for each $e \in \mathsf{A}$, the *atomic action* constant e;
- the *silent step* constant τ;
- the *deadlock* constant δ;
- the binary *alternative composition* operator $+$;
- the binary *sequential composition* operator $\cdot$;
- the binary *parallel composition* operator $\parallel$;
- the binary *left merge* operator $\between$;
- the binary *communication merge* operator $|$;
- for each $H \subseteq \mathsf{A}$, the unary *encapsulation* operator ∂_H;
- for each $I \subseteq \mathsf{A}$, the unary *abstraction* operator τ_I.

We assume that there are infinitely many variables, including x, y, z. Terms are built as usual. We use infix notation for the binary operators.

Let t and t' be closed ACP^τ terms, $e \in \mathsf{A}$, and $H, I \subseteq \mathsf{A}$. Intuitively, the constants and operators to build ACP^τ terms can be explained as follows:

- the process denoted by e first performs atomic action e and next terminates successfully;
- the process denoted by τ performs an unobservable atomic action and next terminates successfully;

Table 5. Axioms of ACP^τ

$x + y = y + x$	A1	$x \cdot \tau = x$	B1
$(x + y) + z = x + (y + z)$	A2	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	B2
$x + x = x$	A3		
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4	$\partial_H(a) = a$ if $a \notin H$	D1
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$\partial_H(a) = \delta$ if $a \in H$	D2
$x + \delta = x$	A6	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\delta \cdot x = \delta$	A7	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4
$x \parallel y = x \between y + y \between x + x \mid y$	CM1	$\tau_I(a) = a$ if $a \notin I$	TI1
$a \between x = a \cdot x$	CM2	$\tau_I(a) = \tau$ if $a \in I$	TI2
$a \cdot x \between y = a \cdot (x \parallel y)$	CM3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI3
$(x + y) \between z = x \between z + y \between z$	CM4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	TI4
$a \cdot x \mid b = (a \mid b) \cdot x$	CM5		
$a \mid b \cdot x = (a \mid b) \cdot x$	CM6	$a \mid b = b \mid a$	C1
$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	CM7	$(a \mid b) \mid c = a \mid (b \mid c)$	C2
$(x + y) \mid z = x \mid z + y \mid z$	CM8	$\delta \mid a = \delta$	C3
$x \mid (y + z) = x \mid y + x \mid z$	CM9	$\tau \mid a = \delta$	C4

- the process denoted by δ can neither perform an atomic action nor terminate successfully;
- the process denoted by $t + t'$ behaves either as the process denoted by t or as the process denoted by t', but not both;
- the process denoted by $t \cdot t'$ first behaves as the process denoted by t and on successful termination of that process it next behaves as the process denoted by t';
- the process denoted by $t \parallel t'$ behaves as the process that proceeds with the processes denoted by t and t' in parallel;
- the process denoted by $t \between t'$ behaves the same as the process denoted by $t \parallel t'$, except that it starts with performing an atomic action of the process denoted by t;
- the process denoted by $t \mid t'$ behaves the same as the process denoted by $t \parallel t'$, except that it starts with performing an atomic action of the process denoted by t and an atomic action of the process denoted by t' synchronously;
- the process denoted by $\partial_H(t)$ behaves the same as the process denoted by t, except that atomic actions from H are blocked;
- the process denoted by $\tau_I(t)$ behaves the same as the process denoted by t, except that atomic actions from I are turned into unobservable atomic actions.

The operators $\between$ and $\mid$ are of an auxiliary nature. They are needed to axiomatize ACP^τ.

The axioms of ACP^τ are given in Table 5. CM2–CM3, CM5–CM7, C1–C4, D1–D4 and TI1–TI4 are actually axiom schemas in which a, b and c stand for

Table 6. RDP, RSP and AIP

$\langle X\|E\rangle = \langle t_X\|E\rangle$	if $X = t_X \in E$	RDP	$\pi_0(a) = \delta$	PR1
$E \Rightarrow X = \langle X\|E\rangle$	if $X \in \mathrm{V}(E)$	RSP	$\pi_{n+1}(a) = a$	PR2
			$\pi_0(a \cdot x) = \delta$	PR3
$\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$		AIP	$\pi_{n+1}(a \cdot x) = a \cdot \pi_n(x)$	PR4
			$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	PR5
			$\pi_n(\tau) = \tau$	PR6
			$\pi_n(\tau \cdot x) = \tau \cdot \pi_n(x)$	PR7

arbitrary constants of ACP^τ, and H and I stand for arbitrary subsets of A. ACP^τ is extended with guarded recursion like BTA.

A *recursive specification* over ACP^τ is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is an ACP^τ term containing only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur in E. Let t be an ACP^τ term without occurrences of abstraction operators containing a variable X. Then an occurrence of X in t is *guarded* if t has a subterm of the form $e \cdot t'$ where $e \in \mathsf{A}$ and t' is a term containing this occurrence of X. Let E be a recursive specification over ACP^τ. Then E is a *guarded recursive specification* if, in each equation $X = t_X \in E$: (i) abstraction operators do not occur in t_X and (ii) all occurrences of variables in t_X are guarded or t_X can be rewritten to such a term using the axioms of ACP^τ in either direction and/or the equations in E except the equation $X = t_X$ from left to right. We are only interested models of ACP^τ in which guarded recursive specifications have unique solutions, such as the models of ACP^τ presented in [2].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ standing for the unique solution of E for X. We write $\langle t_X|E\rangle$ for t_X with, for all $Y \in \mathrm{V}(E)$, all occurrences of Y in t_X replaced by $\langle Y|E\rangle$. The axioms for the constants standing for the unique solution of guarded recursive specifications are RDP and RSP, which are given in Table 6. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary ACP^τ term, and E stands for an arbitrary guarded recursive specification over ACP^τ. Side conditions are added to restrict what X, t_X and E stand for.

Closed terms of ACP^τ extended with constants for solutions of guarded recursive specifications that denote the same process cannot always be proved equal by means of the axioms of ACP^τ together with RDP and RSP. To remedy this, we introduce AIP. AIP is based on the view that two processes are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a process behaves the same as that process, except that it cannot perform any further atomic action after n atomic actions have been performed. AIP is given in Table 6. Here, approximation up to depth n is phrased in terms of a unary *projection* operator π_n. The axioms for these operators are axioms PR1–PR7 in Table 6. PR1–PR7 are actually axiom schemas in which a stands

for arbitrary constants of ACP^τ different from τ and n stands for an arbitrary natural number.

We write ACP^τ+REC for ACP^τ extended with the constants for solutions of guarded recursive specifications, the projection operators, and the axioms RDP, RSP, AIP and PR1–PR7.

The minimal models of ACP^τ+REC are considered its standard models. Henceforth, we restrict ourselves to the minimal models of ACP^τ+REC. We assume that a fixed but arbitrary minimal model $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$ of ACP^τ+REC has been given.

From Section 12, we will sometimes assume that CFAR (Cluster Fair Abstraction Rule) is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. CFAR says that a cluster of silent steps that has exits can be eliminated if all exits are reachable from everywhere in the cluster. A precise formulation of CFAR can be found in [13].

We use the term *process* for the elements from the domain of $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$, and we denote the interpretations of constants and operators in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$ by the constants and operators themselves.

Let P be a process. Then the set of *states* or *subprocesses* of P, written $Sub(P)$, is inductively defined as follows:

- $P \in Sub(P)$;
- if $e \cdot P' \in Sub(P)$, then $P' \in Sub(P)$;
- if $e \cdot P' + P'' \in Sub(P)$, then $P' \in Sub(P)$.

Let P be a process and let $\mathsf{A}' \subseteq \mathsf{A}_\tau$. Then P is *regular over* A' if the following conditions are satisfied:

- $Sub(P)$ is finite;
- for all $P' \in Sub(P)$ and $e \in \mathsf{A}_\tau$, $e \cdot P' \in Sub(P)$ implies $e \in \mathsf{A}'$;
- for all $P', P'' \in Sub(P)$ and $e \in \mathsf{A}_\tau$, $e \cdot P' + P'' \in Sub(P)$ implies $e \in \mathsf{A}'$.

We say that P is *regular* if P is regular over A_τ.

We will make use of the fact that being a regular process over A coincides with being the solution of a finite guarded recursive specification in which the right-hand sides of the recursion equations are linear terms. *Linearity* of terms is inductively defined as follows:

- δ is linear;
- if $e \in \mathsf{A}_\tau$, then e is linear;
- if $e \in \mathsf{A}_\tau$ and X is a variable, then $e \cdot X$ is linear;
- if t and t' are linear, then $t + t'$ is linear.

A *linear recursive specification* over ACP^τ is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over ACP^τ, where each t_X is linear.

Proposition 2. *Let P be a process and let $\mathsf{A}' \subseteq \mathsf{A}$. Then P is regular over A' iff there exists a finite linear recursive specification E over ACP^τ in which only atomic actions from A' occur such that P is the solution of E for some $X \in \mathrm{V}(E)$.*

Table 7. Defining equations for process extraction operation

$\lvert\mathsf{S}\rvert^{\mathrm{c}} = \mathrm{stop}$
$\lvert\mathsf{D}\rvert^{\mathrm{c}} = \mathrm{i} \cdot \delta$
$\lvert T \trianglelefteq \mathsf{tau} \trianglerighteq T'\rvert^{\mathrm{c}} = \mathrm{i} \cdot \mathrm{i} \cdot \lvert T\rvert^{\mathrm{c}}$
$\lvert T \trianglelefteq f.m \trianglerighteq T'\rvert^{\mathrm{c}} = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \lvert T\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot \lvert T'\rvert^{\mathrm{c}})$

Proof. The proof follows the same line as the proof of Proposition 1. □

Remark 1. Proposition 2 is concerned with processes that are regular over A. We can also prove that being a regular process over A_τ coincides with being the solution of a finite linear recursive specification over ACP^τif we assume that the cluster fair abstraction rule [13] holds in the model $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. However, we do not need this more general result.

We will write $\sum_{i\in S} t_i$, where $S = \{i_1, \ldots, i_n\}$ and $t_{i_1}, \ldots, t_{i_n}$ are ACP^τ terms, for $t_{i_1} + \ldots + t_{i_n}$. The convention is that $\sum_{i\in S} t_i$ stands for δ if $S = \emptyset$. We will often write X for $\langle X|E\rangle$ if E is clear from the context. It should be borne in mind that, in such cases, we use X as a constant.

7 Process Extraction

In this section, we use ACP^τ+REC to make mathematically precise which processes are produced by threads.

For that purpose, A and $|$ are taken such that the following conditions are satisfied:[2]

$$\mathsf{A} \supseteq \{\mathrm{s}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{B}\} \cup \{\mathrm{r}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{B}\} \cup \{\mathrm{stop}, \mathrm{i}\}$$

and for all $f \in \mathcal{F}$, $d \in \mathcal{M} \cup \mathbb{B}$, and $e \in \mathsf{A}$:

$$\begin{array}{llll} \mathrm{s}_f(d) \mid \mathrm{r}_f(d) = \mathrm{i}\,, & & & \\ \mathrm{s}_f(d) \mid e = \delta & \mathsf{if}\ e \neq \mathrm{r}_f(d)\,, & \mathrm{stop} \mid e = \delta & \mathsf{if}\ e \neq \mathrm{stop}\,, \\ e \mid \mathrm{r}_f(d) = \delta & \mathsf{if}\ e \neq \mathrm{s}_f(d)\,, & \mathrm{i} \mid e = \delta\,. & \end{array}$$

Actions of the forms $\mathrm{s}_f(d)$ and $\mathrm{r}_f(d)$ are send and receive actions, respectively, stop is an explicit termination action, and i is a concrete internal action.

The *process extraction* operation $|\_|$ assigns a process to each thread. The process extraction operation $|\_|$ is defined by $|T| = \tau_{\{\mathrm{stop}\}}(|T|^{\mathrm{c}})$, where $|\_|^{\mathrm{c}}$ is defined by the equations given in Table 7 (for $f \in \mathcal{F}$ and $m \in \mathcal{M}$).

Let P be a process, T be a thread, and F be an instruction sequence. Then we say that T *produces* P if $\tau \cdot \tau_I(|T|) = \tau \cdot P$ for some $I \subseteq \mathsf{A}$, and we say that F *produces* P if $|F|$ produces P.

Notice that two atomic actions are involved in performing a basic action of the form $f.m$: one for sending a request to process command m to the service

[2] As usual, we will write $\mathbb{B}$ for the set $\{\mathsf{T}, \mathsf{F}\}$.

named f and another for receiving a reply from that service upon completion of the processing. Notice also that, for each thread T, $|T|^{\mathrm{c}}$ is a process that in the event of termination performs a special termination action just before termination. Abstraction from this termination action yields the process denoted by $|T|$.

The process extraction operation preserves the axioms of BTA+REC. Before we make this fully precise, we have a closer look at the axioms of BTA+REC.

A proper axiom is an equation or a conditional equation. In Table 3, we do not find proper axioms. Instead of proper axioms, we find axiom schemas without side conditions and axiom schemas with side conditions. The axioms of BTA+REC are obtained by replacing each axiom schema by all its instances.

Henceforth, we write α^*, where α is a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$, for the unique homomorphic extension of α to terms of BTA+REC. Moreover, we identify $t_1 = t_2$ and $\emptyset \Rightarrow t_1 = t_2$.

Proposition 3. *Let $E \Rightarrow t_1 = t_2$ be an axiom of* BTA+REC*, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $|\alpha^*(t_1)| = |\alpha^*(t_2)|$ if $|\alpha^*(t'_1)| = |\alpha^*(t'_2)|$ for all $t'_1 = t'_2 \in E$.*

Proof. The proof is trivial. □

Remark 2. Proposition 3 would go through if no abstraction of the above-mentioned special termination action was made. Notice further that ACP^{τ} without the silent step constant and the abstraction operator, better known as ACP, would suffice if no abstraction of the special termination action was made.

8 A Simple Protocol for Instruction Stream Processing

In this section and the next section, we consider protocols for instruction stream processing. Before the first protocol is described, an extension of ACP is introduced to simplify the description of the protocols.

The following extension of ACP from [1] will be used: the non-branching conditional operator $:\!\rightarrow$ over $\mathbb{B}$. The expression $b :\!\rightarrow p$, is to be read as `if` b `then` p `else` δ. The axioms for the non-branching conditional operator are

$$\mathsf{T} :\!\rightarrow x = x \quad \text{and} \quad \mathsf{F} :\!\rightarrow x = \delta \,.$$

The protocols concern systems whose main components are an *instruction stream generator* and an *instruction stream execution unit*. The instruction stream generator generates different instruction streams for different threads. This is accomplished by starting it in different states. The general idea of the protocols is that:

- the instruction stream generator generating an instruction stream for a thread $T \trianglelefteq a \trianglerighteq T'$ sends a to the instruction stream execution unit;
- on receipt of a, the instruction stream execution unit gets the execution of a done and sends the reply produced to the instruction stream generator;

– on receipt of the reply, the instruction stream generator proceeds with generating an instruction stream for T if the reply is T and for T' otherwise.

In the case where the thread is S or D, the instruction stream generator sends a special instruction (stop or dead) and the instruction stream execution unit does not send back a reply.

In this section, we consider a very simple protocol for instruction stream processing that makes no effort to keep the execution unit busy without intermission.

We write $\mathcal{I}$ for the set $\mathcal{A} \cup \{\mathsf{stop}, \mathsf{dead}\}$. Elements from $\mathcal{I}$ will loosely be called instructions. The restriction of the domain of $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$ to the regular threads will be denoted by $\mathcal{RT}$.

The functions act, thrt, and thrf defined below give, for each thread T different from S and D, the basic action that T will perform first, the thread with which it will proceed if the reply from the execution environment is T, and the thread with which it will proceed if the reply from the execution environment is F, respectively. The functions $\mathit{act}{:}\mathcal{RT} \to \mathcal{I}$, $\mathit{thrt}{:}\mathcal{RT} \to \mathcal{RT}$, and $\mathit{thrf}{:}\mathcal{RT} \to \mathcal{RT}$ are defined as follows:

$$\begin{array}{lll}
\mathit{act}(\mathsf{S}) = \mathsf{stop}\;, & \mathit{thrt}(\mathsf{S}) = \mathsf{D}\;, & \mathit{thrf}(\mathsf{S}) = \mathsf{D}\;, \\
\mathit{act}(\mathsf{D}) = \mathsf{dead}\;, & \mathit{thrt}(\mathsf{D}) = \mathsf{D}\;, & \mathit{thrf}(\mathsf{D}) = \mathsf{D}\;, \\
\mathit{act}(T \trianglelefteq a \trianglerighteq T') = a\;, & \mathit{thrt}(T \trianglelefteq a \trianglerighteq T') = T\;, & \mathit{thrf}(T \trianglelefteq a \trianglerighteq T') = T'\;.
\end{array}$$

The function nxt^0 defined below is used to distinguish between the execution of a basic action $a \in \mathcal{A}$, which leads to a reply, and the execution of S or D, which leads to termination or inaction. The function $\mathit{nxt}^0 : \mathcal{I} \times \mathcal{RT} \to \mathbb{B}$ is defined as follows:

$$\mathit{nxt}^0(a, T) = \begin{cases} \mathsf{T} & \text{if } \mathit{act}(T) = a \\ \mathsf{F} & \text{if } \mathit{act}(T) \neq a\;. \end{cases}$$

For the purpose of describing the simple protocol outlined above in ACP^τ, A and $\mid$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

$$\begin{array}{l}
\mathsf{A} \supseteq \{\mathrm{s}_i(d) \mid i \in \{1,2\}, d \in \mathcal{I}\} \cup \{\mathrm{r}_i(d) \mid i \in \{1,2\}, d \in \mathcal{I}\} \\
\quad \cup\ \{\mathrm{s}_i(r) \mid i \in \{3,4\}, r \in \mathbb{B}\} \cup \{\mathrm{r}_i(r) \mid i \in \{3,4\}, r \in \mathbb{B}\} \cup \{\mathrm{j}\}
\end{array}$$

and for all $i \in \{1,2\}$, $j \in \{3,4\}$, $d \in \mathcal{I}$, $r \in \mathbb{B}$, and $e \in \mathsf{A}$:

$$\begin{array}{llll}
\mathrm{s}_i(d) \mid \mathrm{r}_i(d) = \mathrm{j}\;, & & \mathrm{s}_j(r) \mid \mathrm{r}_j(r) = \mathrm{j}\;, & \\
\mathrm{s}_i(d) \mid e = \delta & \text{if } e \neq \mathrm{r}_i(d)\;, & \mathrm{s}_j(r) \mid e = \delta & \text{if } e \neq \mathrm{r}_j(r)\;, \\
e \mid \mathrm{r}_i(d) = \delta & \text{if } e \neq \mathrm{s}_i(d)\;, & e \mid \mathrm{r}_j(r) = \delta & \text{if } e \neq \mathrm{s}_j(r)\;,
\end{array}$$

$$\mathrm{j} \mid e = \delta\;.$$

Notice that the set $\mathbb{B}$ is the set of replies.

Let $T \in \mathcal{RT}$. Then the process representing the simple protocol for instruction stream processing with regard to thread T is described by

$$\partial_H(\mathit{ISG}^0_T \parallel \mathit{IMTC}^0 \parallel \mathit{RTC}^0 \parallel \mathit{ISEU}^0)\;,$$

where the process ISG_T^0 is recursively specified by the following equation:

$$\begin{aligned} ISG_T^0 = \sum_{f.m \in \mathcal{A}} & nxt^0(f.m, T) :\rightarrow \\ & \mathrm{s}_1(f.m) \cdot (\mathrm{r}_4(\mathsf{T}) \cdot ISG^0_{thrt(T)} + \mathrm{r}_4(\mathsf{F}) \cdot ISG^0_{thrf(T)}) \\ + \; & nxt^0(\mathsf{stop}, T) :\rightarrow \mathrm{s}_1(\mathsf{stop}) + nxt^0(\mathsf{dead}, T) :\rightarrow \mathrm{s}_1(\mathsf{dead}) \;, \end{aligned}$$

the process $IMTC^0$ is recursively specified by the following equation:

$$IMTC^0 = \sum_{d \in \mathcal{I}} \mathrm{r}_1(d) \cdot \mathrm{s}_2(d) \cdot IMTC^0 \;,$$

the process RTC^0 is recursively specified by the following equation:

$$RTC^0 = \sum_{r \in \mathbb{B}} \mathrm{r}_3(r) \cdot \mathrm{s}_4(r) \cdot RTC^0 \;,$$

the process $ISEU^0$ is recursively specified by the following equation:

$$\begin{aligned} ISEU^0 = & \sum_{f.m \in \mathcal{A}} \mathrm{r}_2(f.m) \cdot \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \mathrm{s}_3(\mathsf{T}) + \mathrm{r}_f(\mathsf{F}) \cdot \mathrm{s}_3(\mathsf{F})) \cdot ISEU^0 \\ & + \mathrm{r}_2(\mathsf{stop}) + \mathrm{r}_2(\mathsf{dead}) \cdot \mathrm{i} \cdot \delta \end{aligned}$$

and

$$\begin{aligned} H \;=\; & \{\mathrm{s}_i(d) \mid i \in \{1,2\}, d \in \mathcal{I}\} \cup \{\mathrm{r}_i(d) \mid i \in \{1,2\}, d \in \mathcal{I}\} \\ \cup \; & \{\mathrm{s}_i(r) \mid i \in \{3,4\}, r \in \mathbb{B}\} \cup \{\mathrm{r}_i(r) \mid i \in \{3,4\}, r \in \mathbb{B}\} \;. \end{aligned}$$

ISG_T^0 is the instruction stream generator for thread T, $IMTC^0$ is the transmission channel for messages containing instructions, RTC^0 is the transmission channel for replies, and $ISEU^0$ is the instruction stream execution unit.

If we abstract from all communications via the transmission channels, then the process denoted by $\partial_H(ISG_T^0 \parallel IMTC^0 \parallel RTC^0 \parallel ISEU^0)$ and the process $|T|$ are equal modulo an initial silent step.

Theorem 1. *For each $T \in \mathcal{RT}$, $\tau \cdot \tau_{\{\mathrm{j}\}}(\partial_H(ISG_T^0 \parallel IMTC^0 \parallel RTC^0 \parallel ISEU^0))$ denotes the process $\tau \cdot |T|$.*

Proof. Let $T \in \mathcal{RT}$. Moreover, let E be a linear recursive specification over ACP^τ with $X \in \mathrm{V}(E)$ such that $|T|$ is the solution of E for X in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. By Proposition 2 and the definition of the process extraction operation, it is sufficient to prove that

$$\tau \cdot \tau_{\{\mathrm{j}\}}(\partial_H(ISG_T^0 \parallel IMTC^0 \parallel RTC^0 \parallel ISEU^0)) = \tau \cdot \langle X|E\rangle \;.$$

By AIP, it is sufficient to prove that for all $n \geq 0$:

$$\pi_n(\tau \cdot \tau_{\{\mathrm{j}\}}(\partial_H(ISG_T^0 \parallel IMTC^0 \parallel RTC^0 \parallel ISEU^0))) = \pi_n(\tau \cdot \langle X|E\rangle) \;.$$

This is easily proved by induction on n and in the inductive step by case distinction on the structure of T, using the axioms of ACP^τ and RDP and in addition the fact that $|T'| \in Sub(|T|)$ for all $T' \in Res(T)$ and the fact that there exists an bijection between $Sub(|T|)$ and $\mathrm{V}(E)$. □

9 A More Complex Protocol

In this section, we consider a more complex protocol for instruction stream processing that makes an effort to keep the execution unit busy without intermission.

The specifics of the more complex protocol considered here are that:

- the instruction stream generator may run ahead of the instruction stream execution unit by not waiting for the receipt of the replies resulting from the execution of instructions that it has sent earlier;
- to ensure that the instruction stream execution unit can handle the run-ahead, each instruction sent by the instruction stream generator is accompanied with the sequence of replies after which the instruction must be executed;
- to correct for replies that have not yet reached the instruction stream generator, each instruction sent is also accompanied with the number of replies received since the last sending of an instruction.

We write $\mathbb{B}^{\leq n}$, where $n \in \mathbb{N}$, for the set $\{u \in \mathbb{B}^* \mid \mathit{len}(u) \leq n\}$.[3]

It is assumed that a natural number ℓ has been given. The number ℓ is taken for the maximal number of steps that the instruction stream generator may run ahead of the instruction stream execution unit.

The set $\mathcal{IM}$ of *instruction messages* is defined as follows:

$$\mathcal{IM} = [0, \ell] \times \mathbb{B}^{\leq \ell} \times \mathcal{I} \ .$$

In an instruction message $(n, u, a) \in \mathcal{IM}$:

- n is the number of replies that are acknowledged by the message;
- u is the sequence of replies after which the instruction that is part of the message must be executed;
- a is the instruction that is part of the message.

The instruction stream generator sends instruction messages via an instruction message transmission channel to the instruction stream execution unit. We refer to a succession of transmitted instruction messages as an *instruction stream*. An instruction stream is dynamic by nature, in contradistinction with an instruction sequence.

The set $\mathcal{S}_{\text{ISG}}$ of *instruction stream generator states* is defined as follows:

$$\mathcal{S}_{\text{ISG}} = [0, \ell] \times \mathcal{P}(\mathbb{B}^{\leq \ell+1} \times \mathcal{RT}) \ .$$

In an instruction stream generator state $(n, R) \in \mathcal{S}_{\text{ISG}}$:

- n is the number of replies that has been received by the instruction stream generator since the last acknowledgement of received replies;

[3] As usual, we write D^* for the set of all finite sequences with elements from set D and $\mathit{len}(\sigma)$ for the length of finite sequence σ. Moreover, we write ϵ for the empty sequence, d for the sequence having d as sole element, $\sigma\sigma'$ for the concatenation of finite sequences σ and σ', and $\mathit{tl}(\sigma)$ for the tail of finite sequence σ.

– in each $(u, T) \in R$, u is the sequence of replies after which the thread T must be performed.

The functions *updpm* and *updcr* defined below are used to model the updates of the instruction stream generator state on producing a message and consuming a reply, respectively. The function $updpm : (\mathbb{B}^{\leq \ell} \times \mathcal{RT}) \times \mathcal{S}_{\text{ISG}} \to \mathcal{S}_{\text{ISG}}$ is defined as follows:

$$updpm((u, T), (n, R)) = \begin{cases} (0, (R \setminus \{(u, T)\}) \cup \{(u\mathsf{T}, thrt(T)), (u\mathsf{F}, thrf(T))\}) & \text{if } act(T) \notin \{\mathsf{S}, \mathsf{D}\} \\ (0, (R \setminus \{(u, T)\})) & \text{if } act(T) \in \{\mathsf{S}, \mathsf{D}\} \end{cases} .$$

The function $updcr : \mathbb{B} \times \mathcal{S}_{\text{ISG}} \to \mathcal{S}_{\text{ISG}}$ is defined as follows:

$$updcr(r, (n, R)) = (n + 1, \{(u, T) \mid (ru, T) \in R\}) .$$

The function *sel* defined below is used to model the selection of the sequence of replies and the instruction that will be part of the next message produced by the instruction stream generator. The function $sel : \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{RT}) \to \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{RT})$ is defined as follows:

$$sel(R) = \{(u, T) \in R \mid \forall (v, T') \in R \bullet len(u) \leq len(v) \wedge len(u) \leq \ell\} .$$

Notice that $(u, T) \in sel(R)$ and $(v, T') \in R$ only if $u \leq v$. By that depth-first run-ahead is excluded. It happens that the performance of the protocol may change considerably if the function *sel* is replaced by another function.

The set $\mathcal{S}_{\text{ISEU}}$ of *instruction stream execution unit states* is defined as follows:

$$\mathcal{S}_{\text{ISEU}} = [0, \ell] \times \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{I}) .$$

In an instruction stream execution unit state $(n, S) \in \mathcal{S}_{\text{ISEU}}$:

– n is the number of replies for which the instruction stream execution unit still has to receive an acknowledgement;
– in each $(u, a) \in S$, u is the sequence of replies after which the instruction a must be executed.

The functions *updcm* and *updpr* defined below are used to model the updates of the instruction stream execution unit state on producing a reply and consuming a message, respectively. The function $updcm : \mathcal{IM} \times \mathcal{S}_{\text{ISEU}} \to \mathcal{S}_{\text{ISEU}}$ is defined as follows:

$$updcm((k, u, a), (n, S)) = (n - k, S \cup \{(tl^{n-k}(u), a)\}) .^{4}$$

The function $updpr : \mathbb{B} \times \mathcal{S}_{\text{ISEU}} \to \mathcal{S}_{\text{ISEU}}$ is defined as follows:

$$updpr(r, (n, S)) = (n + 1, \{(u, a) \mid (ru, a) \in S\}) .$$

The function *nxt* defined below is used to distinguish between the execution of a basic action $a \in \mathcal{A}$, which leads to a reply, and the execution of S or D, which

[4] $tl^n(u)$ is defined by induction on n as usual: $tl^0(u) = u$ and $tl^{n+1}(u) = tl(tl^{n+1}(u))$.

leads to termination or inaction. The function $nxt{:}\mathcal{I}\times\mathcal{P}(\mathbb{B}^{\leq\ell}\times\mathcal{I})\to\mathbb{B}$ is defined as follows:

$$nxt(a,S)=\begin{cases}\mathsf{T} & \text{if } (\epsilon,a)\in S\\ \mathsf{F} & \text{if } (\epsilon,a)\notin S\ .\end{cases}$$

The instruction stream execution unit sends replies via a reply transmission channel to the instruction stream generator. We refer to a succession of transmitted replies as a *reply stream*.

For the purpose of describing the transmission protocol in ACP^τ, A and $|$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

$$\begin{aligned}\mathsf{A}\ \supseteq\ &\{\mathrm{s}_i(d)\mid i\in\{1,2\},d\in\mathcal{IM}\}\cup\{\mathrm{r}_i(d)\mid i\in\{1,2\},d\in\mathcal{IM}\}\\ \cup\ &\{\mathrm{s}_i(r)\mid i\in\{3,4\},r\in\mathbb{B}\}\cup\{\mathrm{r}_i(r)\mid i\in\{3,4\},r\in\mathbb{B}\}\cup\{\mathrm{j}\}\end{aligned}$$

and for all $i\in\{1,2\}$, $j\in\{3,4\}$, $d\in\mathcal{IM}$, $r\in\mathbb{B}$, and $e\in\mathsf{A}$:

$$\begin{aligned}&\mathrm{s}_i(d)\mid\mathrm{r}_i(d)=\mathrm{j}\ , && && \mathrm{s}_j(r)\mid\mathrm{r}_j(r)=\mathrm{j}\ , && \\ &\mathrm{s}_i(d)\mid e=\delta && \text{if } e\neq\mathrm{r}_i(d)\ , && \mathrm{s}_j(r)\mid e=\delta && \text{if } e\neq\mathrm{r}_j(r)\ ,\\ &e\mid\mathrm{r}_i(d)=\delta && \text{if } e\neq\mathrm{s}_i(d)\ , && e\mid\mathrm{r}_j(r)=\delta && \text{if } e\neq\mathrm{s}_j(r)\ ,\end{aligned}$$

$$\mathrm{j}\mid e=\delta\ .$$

Let $T\in\mathcal{RT}$. Then the process representing the more complex protocol for instruction stream processing with regard to thread T is described by

$$\partial_H(ISG_T\parallel IMTC\parallel RTC\parallel ISEU)\ ,$$

where the process ISG_T is recursively specified by the following equations:

$$ISG_T = ISG'_{(0,\{(\epsilon,T)\})}\ ,$$

$$\begin{aligned}ISG'_{(n,R)} = &\sum_{(u,T)\in sel(R)}\mathrm{s}_1((n,u,act(T)))\cdot ISG'_{updpm((u,T),(n,R))}\\ &+\sum_{r\in\mathbb{B}}\mathrm{r}_4(r)\cdot ISG'_{updcr(r,(n,R))}\end{aligned}$$

(for every $(n,R)\in\mathcal{S}_{\mathrm{ISG}}$ with $R\neq\emptyset$) ,

$$ISG'_{(n,\emptyset)} = \mathrm{j}$$

(for every $(n,\emptyset)\in\mathcal{S}_{\mathrm{ISG}}$) ,

the process $IMTC$ is recursively specified by the following equation:

$$IMTC=\sum_{d\in\mathcal{IM}}\mathrm{r}_1(d)\cdot\mathrm{s}_2(d)\cdot IMTC\ ,$$

the process RTC is recursively specified by the following equation:

$$RTC=\sum_{r\in\mathbb{B}}\mathrm{r}_3(r)\cdot\mathrm{s}_4(r)\cdot RTC\ ,$$

the process $ISEU$ is recursively specified by the following equations:

$$\begin{aligned}
ISEU &= ISEU'_{(0,\emptyset)} \;,\\
ISEU'_{(n,S)} &= \sum_{d\in\mathcal{IM}} \mathrm{r}_2(d)\cdot ISEU'_{updcm(d,(n,S))}\\
&\quad + \sum_{f.m\in\mathcal{A}} nxt(f.m,S) :\rightarrow \mathrm{s}_f(m)\cdot ISEU''_{(n,S)}\\
&\quad + nxt(\mathsf{stop},S) :\rightarrow \mathrm{j} + nxt(\mathsf{dead},S) :\rightarrow \mathrm{i}\cdot\delta\\
&\text{(for every } (n,S)\in\mathcal{S}_{\mathrm{ISEU}})\;,\\
ISEU''_{(n,S)} &= \sum_{r\in\mathbb{B}} \mathrm{r}_f(r)\cdot \mathrm{s}_3(r)\cdot ISEU'_{updpr(r,(n,S))}\\
&\quad + \sum_{d\in\mathcal{IM}} \mathrm{r}_2(d)\cdot ISEU''_{updcm(d,(n,S))}\\
&\text{(for every } (n,S)\in\mathcal{S}_{\mathrm{ISEU}})\;,
\end{aligned}$$

and

$$\begin{aligned}
H \;=\; &\{\mathrm{s}_i(d)\mid i\in\{1,2\}, d\in\mathcal{IM}\}\cup\{\mathrm{r}_i(d)\mid i\in\{1,2\}, d\in\mathcal{IM}\}\\
\cup\; &\{\mathrm{s}_i(r)\mid i\in\{3,4\}, r\in\mathbb{B}\}\cup\{\mathrm{r}_i(r)\mid i\in\{3,4\}, r\in\mathbb{B}\}\;.
\end{aligned}$$

ISG_T is the instruction stream generator for thread T, $IMTC$ is the transmission channel for instruction messages, RTC is the transmission channel for replies, and $ISEU$ is the instruction stream execution unit.

The protocol described above has been designed such that, for each $T\in\mathcal{RT}$, $\tau\cdot\tau_{\{\mathrm{j}\}}(\partial_H(ISG_T \parallel IMTC \parallel RTC \parallel ISEU))$ denotes the process $\tau\cdot|T|$. We refrain from presenting a proof of the claim that the protocol satisfies this because this paper is first and foremost a conceptual paper and the proof is straightforward but tedious.

The transmission channels $IMTC$ and RTC can keep one instruction message and one reply, respectively. The protocol has been designed in such a way that the protocol will also work properly if these channels are replaced by channels with larger capacity and even by channels with unbounded capacity.

10 Adaptations of the Protocol

In this section, we discuss some conceivable adaptations of the protocol described in Section 9.

Consider the case where, for each instruction, it is known what the probability is with which its execution leads to the reply T. This might give reason to adapt the protocol described in Section 9. Suppose that the instruction stream generator states do not only keep the sequences of replies after which threads must be performed, but also the sequences of instructions involved in producing those sequences of replies. Then the probability with which the sequences of replies will happen can be calculated and several conceivable adaptations of the protocol to this probabilistic knowledge are possible by mere changes in the

selection of the sequence of replies and the instruction that will be part of the next instruction message produced by the instruction stream generator. Among those adaptations are:

- restricting the instruction messages that are produced ahead to the ones where the sequence of replies after which the instruction must be executed will happen with a probability ≥ 0.50, but sticking to breadth-first run-ahead;
- restricting the instruction messages that are produced ahead to the ones where the sequence of replies after which the instruction must be executed will happen with a probability ≥ 0.95, but not sticking to breadth-first run-ahead.

Regular threads can be represented in such a way that it is effectively decidable whether the two threads with which a thread may proceed after performing its first action are identical. Consider the case where threads are represented in the instruction stream generator states in such a way. Then the protocol can be adapted such that no duplication of instruction messages takes place in the cases where the two threads with which a thread possibly proceeds after performing its first action are identical. This can be accomplished by using sequences of elements from $\mathbb{B} \cup \{*\}$, instead of sequences of elements from $\mathbb{B}$, in instruction messages, instruction stream generator states, and instruction stream execution unit states. The occurrence of $*$ at position i in a sequence indicates that the ith reply may be either T or F. The impact of this change on the updates of instruction stream generator states and instruction stream execution unit states is minor.

11 Alternative Choice Instructions

Process algebras such as ACP, CCS and CSP are considered relevant to computer science, as is witnesses by the extent of the work on them in theoretical computer science. This means that there must be programmed systems whose behaviours are taken for processes as considered in such a process algebra. In coming sections, we will establish results concerning the processes as considered in ACP that can be produced by programs under execution, starting from the perception of a program as an instruction sequence.

For the purpose of producing processes as considered in ACP, we need a version of PGA with special basic instructions. Recall that, in PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. In the coming sections, we will make use a version of PGA in which the following additional assumptions relating to $\mathfrak{A}$ are made:

- a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given;
- a fixed but arbitrary set $\mathcal{M}$ of *methods* has been given;
- a fixed but arbitrary set $\mathcal{AA}$ of *atomic actions*, with $\mathrm{t} \notin \mathcal{AA}$, has been given;
- $\mathfrak{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\} \cup \{\mathsf{ac}(e_1, e_2) \mid e_1, e_2 \in \mathcal{AA} \cup \{\mathrm{t}\}\}$.

Table 8. Additional defining equation for process extraction operation

$$\overline{|T \trianglelefteq \mathsf{ac}(e,e') \trianglerighteq T'|^{\mathrm{c}} = e \cdot |T|^{\mathrm{c}} + e' \cdot |T'|^{\mathrm{c}}}$$

On execution of a basic instruction $\mathsf{ac}(e_1, e_2)$, first a non-deterministic choice between the atomic actions e_1 and e_2 is made and then the chosen atomic action is performed. The reply T is produced if e_1 is performed and the reply F is produced if e_2 is performed. Basic instructions of this kind are material to produce all regular processes by means of instruction sequences. A basic instruction of the form $\mathsf{ac}(e_1, e_2)$ is called a *alternative choice instruction*. Henceforth, we will write $\mathrm{PGA}_{\mathrm{ac}}$ for the version of PGA with alternative choice instructions.

The intuition concerning alternative choice instructions given above will be made fully precise at the end of this section, using ACP or rather ACP^{τ}. It will not be made fully precise using an extension of BTA because it is considered a basic property of threads that they are deterministic behaviours.

Recall that we make use of a version of BTA in which $\mathcal{A} = \mathfrak{A}$. A basic action of the form $\mathsf{ac}(e_1, e_2)$ is called a *alternative choice action*. Henceforth, we will write $\mathrm{BTA}^{\mathrm{ac}}$ for the version of BTA with alternative choice actions.

For the purpose of making precise what processes are produced by the threads denoted by closed terms of $\mathrm{BTA}^{\mathrm{ac}}$+REC, A and $|$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

$$\mathsf{A} \supseteq \mathcal{A}\mathcal{A} \cup \{\mathrm{t}\}$$

and for all $e, e' \in \mathsf{A}$:

$$e' \mid e = \delta \ \ \mathsf{if}\ e' \in \mathcal{A}\mathcal{A} \cup \{\mathrm{t}\}\ .$$

The process extraction operation for $\mathrm{BTA}^{\mathrm{ac}}$ has as defining equations the equations given in Table 7 and in addition the equation given in Table 8.

Proposition 3 goes through for $\mathrm{BTA}^{\mathrm{ac}}$.

12 Instruction Sequence Producible Processes

It follows immediately from the definitions of the thread extraction and process extraction operations that the instruction sequences considered in PGA produce regular processes. The question is whether all regular processs are producible by these instruction sequences. In this section, we show that all regular processes can be produced by the instruction sequences with alternative choice instructions.

We will make use of the fact that all regular threads over $\mathcal{A}$ can be produced by the single-pass instruction sequences considered in PGA.

Proposition 4. *For each thread T that is regular over $\mathcal{A}$, there exists a* PGA *instruction sequence F such that F produces T, i.e. $|F| = T$.*

Proof. This proposition generalizes one direction of Proposition 2 from [18] from the projective limit model of BTA to an arbitrary minimal model of BTA. However, the proof of that proposition is applicable to any minimal model of BTA. □

All regular processes over $\mathcal{AA}$ can be produced by the instruction sequences considered in $\mathrm{PGA}_{\mathrm{ac}}$.

Theorem 2. *Assume that CFAR is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists an instruction sequence F in which only basic instructions of the form $\mathsf{ac}(e,\mathrm{t})$ occur such that F produces P, i.e. $\tau\cdot\tau_{\{\mathrm{t}\}}(||F||)=\tau\cdot P$.*

Proof. By Propositions 1, 2 and 4, it is sufficient to show that, for each finite linear recursive specification E over ACP^τ in which only atomic actions from $\mathcal{AA}$ occur, there exists a finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{ac}}$ in which only basic actions of the form $\mathsf{ac}(e,\mathrm{t})$ occur such that $\tau\cdot\langle X|E\rangle=\tau\cdot\tau_{\{\mathrm{t}\}}(|\langle X|E'\rangle|)$ for all $X\in\mathrm{V}(E)$.

Take the finite linear recursive specification E over ACP^τ that consists of the recursion equations

$$X_i = e_{i1}\cdot X_{i1}+\ldots+e_{ik_i}\cdot X_{ik_i}+e'_{i1}+\ldots+e'_{il_i}\;,$$

where $e_{i1},\ldots,e_{ik_i},e'_{i1},\ldots,e'_{il_i}\in\mathcal{AA}$, for $i\in\{1,\ldots n\}$. Then construct the finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{ac}}$ that consists of the recursion equations

$$\begin{aligned}X_i = X_{i1}\trianglelefteq\mathsf{ac}(e_{i1},\mathrm{t})\trianglerighteq(\ldots(X_{ik_i}\trianglelefteq\mathsf{ac}(e_{ik_i},\mathrm{t})\trianglerighteq\\ (\mathsf{S}\trianglelefteq\mathsf{ac}(e'_{i1},\mathrm{t})\trianglerighteq(\ldots(\mathsf{S}\trianglelefteq\mathsf{ac}(e'_{il_i},\mathrm{t})\trianglerighteq X_i)\ldots)))\ldots)\end{aligned}$$

for $i\in\{1,\ldots n\}$; and the finite linear recursive specification E'' over ACP^τ that consists of the recursion equations

$$\begin{aligned}X_i &= e_{i1}\cdot X_{i1}+\mathrm{t}\cdot Y_{i2}\;, & Z_{i1} &= e'_{i1}+\mathrm{t}\cdot Z_{i2}\;,\\ Y_{i2} &= e_{i2}\cdot X_{i2}+\mathrm{t}\cdot Y_{i3}\;, & Z_{i2} &= e'_{i2}+\mathrm{t}\cdot Z_{i3}\;,\\ &\vdots & &\vdots\\ Y_{ik_i} &= e_{ik_i}\cdot X_{ik_i}+\mathrm{t}\cdot Z_{i1}\;, & Z_{il_i} &= e'_{il_i}+\mathrm{t}\cdot X_i\;,\end{aligned}$$

where $Y_{i2},\ldots,Y_{ik_i},Z_{i1},\ldots,Z_{il_i}$ are fresh variables, for $i\in\{1,\ldots n\}$. It follows immediately from the definition of the process extraction operation that $|\langle X|E'\rangle|=\langle X|E''\rangle$ for all $X\in\mathrm{V}(E)$. Moreover, it follows from CFAR that $\tau\cdot\langle X|E\rangle=\tau\cdot\tau_{\{\mathrm{t}\}}(\langle X|E''\rangle)$ for all $X\in\mathrm{V}(E)$. Hence, $\tau\cdot\langle X|E\rangle=\tau\cdot\tau_{\{\mathrm{t}\}}(|\langle X|E'\rangle|)$ for all $X\in\mathrm{V}(E)$. □

Remark 3. Theorem 2 with "$\tau\cdot\tau_{\{\mathrm{t}\}}(||F||)=\tau\cdot P$" replaced by "$||F||=P$" can be established if PGA is extended with multiple-reply test instructions, see [7]. In that case, the assumption that CFAR is valid is superfluous.

13 Services and Use Operators

An instruction sequence under execution may make use of services. That is, certain instructions may be executed for the purpose of having the behaviour produced by the instruction sequence affected by a service that takes those instructions as commands to be processed. Likewise, a thread may perform certain actions for the purpose of having itself affected by a service that takes those actions as commands to be processed. The processing of an action may involve a change of state of the service and at completion of the processing of the action the service returns a reply value to the thread. The reply value determines how the thread proceeds. The use operators can be used in combination with the thread extraction operation from Section 4 to describe the behaviour produced by instruction sequences that make use of services. In this section, we first review the use operators, which are concerned with threads making such use of services, and then extend the process extraction operation to the use operators.

A *service* H consists of

- a set S of *states*;
- an *effect* function $\mathit{eff} : \mathcal{M} \times S \to S$;
- a *yield* function $\mathit{yld} : \mathcal{M} \times S \to \mathbb{B} \cup \{\mathsf{B}\}$;
- an *initial state* $s_0 \in S$;

satisfying the following condition:

$$\forall m \in \mathcal{M}, s \in S \bullet (\mathit{yld}(m, s) = \mathsf{B} \Rightarrow \forall m' \in \mathcal{M} \bullet \mathit{yld}(m', \mathit{eff}(m, s)) = \mathsf{B}) \ .$$

The set S contains the states in which the service may be, and the functions *eff* and *yld* give, for each method m and state s, the state and reply, respectively, that result from processing m in state s. By the condition imposed on services, once the service has returned B as reply, it keeps returning B as reply.

Let $H = (S, \mathit{eff}, \mathit{yld}, s_0)$ be a service and let $m \in \mathcal{M}$. Then the *derived service* of H after processing m, written $\frac{\partial}{\partial m} H$, is the service $(S, \mathit{eff}, \mathit{yld}, \mathit{eff}(m, s_0))$; and the *reply* of H after processing m, written $H(m)$, is $\mathit{yld}(m, s_0)$.

When a thread makes a request to the service to process m:

- if $H(m) \neq \mathsf{B}$, then the request is accepted, the reply is $H(m)$, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(m) = \mathsf{B}$, then the request is rejected and the service proceeds as a service that rejects any request.

We introduce the sort $\mathbf{S}$ of *services*. However, we will not introduce constants and operators to build terms of this sort. The sort $\mathbf{S}$, standing for the set of all services, is considered a parameter of the extension of BTA being presented. Moreover, we introduce, for each $f \in \mathcal{F}$, the binary *use* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. The axioms for these operators are given in Table 9. Intuitively, $T /_f H$ is the thread that results from processing all actions performed by thread T that are of the form $f.m$ by service H. When a basic action of the form $f.m$ performed by thread T is processed by service H, it is turned into the basic action tau and

Table 9. Axioms for use operators

$\mathsf{S} \mathbin{/_f} H = \mathsf{S}$		U1
$\mathsf{D} \mathbin{/_f} H = \mathsf{D}$		U2
$(x \trianglelefteq \mathsf{tau} \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq \mathsf{tau} \trianglerighteq (y \mathbin{/_f} H)$		U3
$(x \trianglelefteq g.m \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq g.m \trianglerighteq (y \mathbin{/_f} H)$	if $f \neq g$	U4
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{T}$	U5
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (y \mathbin{/_f} \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{F}$	U6
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{D}$	if $H(m) = \mathsf{B}$	U7
$(x \trianglelefteq \mathsf{ac}(e_1, e_2) \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq \mathsf{ac}(e_1, e_2) \trianglerighteq (y \mathbin{/_f} H)$		U8
$\pi_n(x \mathbin{/_f} H) = \pi_n(\pi_n(x) \mathbin{/_f} H)$		U9

postconditional composition is removed in favour of basic action prefixing on the basis of the reply value produced.

We add the use operators to $\mathrm{PGA}_{\mathrm{ac}}$ as well. We will only use the extension in combination with the thread extraction operation $|\_|$ and define $|F \mathbin{/_f} H| = |F| \mathbin{/_f} H$. Hence, $|F \mathbin{/_f} H|$ denotes the thread produced by F if F makes use of H. If H is a service such as an unbounded counter, an unbounded stack or a Turing tape, then a non-regular thread may be produced.

In order to extend the process extraction operation to the use operators, we need an extension of ACP^τ with action renaming operators ρ_h, where $h{:}\mathsf{A}_\tau \to \mathsf{A}_\tau$ such that $h(\tau) = \tau$. The axioms for action renaming are given in [13]. Intuitively, $\rho_h(P)$ behaves as P with each atomic action replaced according to h. We write $\rho_{e' \mapsto e''}$ for the renaming operator ρ_h with h defined by $h(e') = e''$ and $h(e) = e$ if $e \neq e'$.

For the purpose of extending the process extraction operation to the use operators, A and $|$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, with everywhere $\mathbb{B}$ replaced by $\mathbb{B} \cup \{\mathsf{B}\}$, and the conditions mentioned at the end of Section 11, the following conditions are satisfied:

$$\mathsf{A} \supseteq \{\mathrm{s}_{\mathrm{serv}}(r) \mid r \in \mathbb{B} \cup \{\mathsf{B}\}\} \cup \{\mathrm{r}_{\mathrm{serv}}(m) \mid m \in \mathcal{M}\} \cup \{\mathrm{stop}^*\}$$

and for all $e \in \mathsf{A}$, $m \in \mathcal{M}$, and $r \in \mathbb{B} \cup \{\mathsf{B}\}$:

$$\begin{aligned} &\mathrm{s}_{\mathrm{serv}}(r) \mid e = \delta \;, && \mathrm{stop} \mid \mathrm{stop} = \mathrm{stop}^* \;, \\ &e \mid \mathrm{r}_{\mathrm{serv}}(m) = \delta \;, && \mathrm{stop}^* \mid e = \delta \;. \end{aligned}$$

We also need to define a set $A_f \subseteq \mathsf{A}$ and a function $h_f : \mathsf{A}_\tau \to \mathsf{A}_\tau$ for each $f \in \mathcal{F}$:

$$A_f = \{\mathrm{s}_f(d) \mid d \in \mathcal{M} \cup \mathbb{B} \cup \{\mathsf{B}\}\} \cup \{\mathrm{r}_f(d) \mid d \in \mathcal{M} \cup \mathbb{B} \cup \{\mathsf{B}\}\} \;;$$

for all $e \in \mathsf{A}_\tau$, $m \in \mathcal{M}$ and $r \in \mathbb{B} \cup \{\mathsf{B}\}$:

$$\begin{aligned} &h_f(\mathrm{s}_{\mathrm{serv}}(r)) = \mathrm{s}_f(r) \;, \\ &h_f(\mathrm{r}_{\mathrm{serv}}(m)) = \mathrm{r}_f(m) \;, \\ &h_f(e) = e && \text{if } \textstyle\bigwedge_{r' \in \mathbb{N}} e \neq \mathrm{s}_{\mathrm{serv}}(r') \wedge \bigwedge_{m' \in \mathcal{M}} e \neq \mathrm{r}_{\mathrm{serv}}(m') \;. \end{aligned}$$

Table 10. Adapted and additional defining equations for process extraction operation

$$\overline{|T \trianglelefteq f.m \trianglerighteq T'|^{\mathrm{c}} = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot |T|^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot |T'|^{\mathrm{c}} + \mathrm{r}_f(\mathsf{B}) \cdot \mathrm{i} \cdot \delta)}$$
$$\underline{|T /_f H|^{\mathrm{c}} = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(|T|^{\mathrm{c}} \parallel \rho_{h_f}(|H|^{\mathrm{c}}))))}$$

To extend the process extraction operation to the use operators, the defining equation concerning the postconditional composition operators has to be adapted and a new defining equation concerning the use operators has to be added. These two equations are given in Table 10, where $|H|^{\mathrm{c}}$ is the solution of

$$\{X_{H'} = \sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(m)) \cdot X_{\frac{\partial}{\partial m} H'} + \mathrm{stop} \mid H' \in \Delta(H)\}$$

for X_H, where $\Delta(H)$ is inductively defined as follows:

- $H \in \Delta(H)$;
- if $m \in \mathcal{M}$ and $H' \in \Delta(H)$, then $\frac{\partial}{\partial m} H' \in \Delta(H)$.

The extended process extraction operation preserves the axioms for the use operators. Owing to the presence of axiom schemas with semantic side conditions in Table 9, the axioms for the use operators include proper axioms, which are all of the form $t_1 = t_2$, and axioms that have a semantic side condition, which are all of the form $t_1 = t_2$ if $H(m) = r$. By that, the precise formulation of the preservation result is somewhat complicated. On the other hand

Proposition 5.

1. *Let $t_1 = t_2$ be a proper axiom for the use operators, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $|\alpha^*(t_1)| = |\alpha^*(t_2)|$.*
2. *Let $t_1 = t_2$ if $H(m) = r$ be an axiom with semantic side condition for the use operators, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $|\alpha^*(t_1)| = |\alpha^*(t_2)|$ if $H(m) = r$.*

Proof. The proof is straightforward. We sketch the proof for axiom U5. By the definition of the process extraction operation, it is sufficient to show that $|(T \trianglelefteq f.m \trianglerighteq T') /_f H|^{\mathrm{c}} = |\mathsf{tau} \circ (T /_f \frac{\partial}{\partial m} H)|^{\mathrm{c}}$ if $H(m) = \mathsf{T}$. In outline, this goes as follows:

$$\begin{aligned}
&|(T \trianglelefteq f.m \trianglerighteq T') /_f H|^{\mathrm{c}} \\
&= \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}} \\
&\quad (\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(\mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot |T|^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot |T'|^{\mathrm{c}} + \mathrm{r}_f(\mathsf{B}) \cdot \mathrm{i} \cdot \delta) \parallel \rho_{h_f}(|H|^{\mathrm{c}})))) \\
&= \mathrm{i} \cdot \mathrm{i} \cdot \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(|T|^{\mathrm{c}} \parallel \rho_{h_f}(|\tfrac{\partial}{\partial m} H|^{\mathrm{c}})))) \\
&= |\mathsf{tau} \circ (T /_f \tfrac{\partial}{\partial m} H)|^{\mathrm{c}} \,.
\end{aligned}$$

In the first and third step, we apply defining equations of $|\_|^{\mathrm{c}}$. In the second step, we apply axioms of ACP$^\tau$+REC with action renaming, and use that $H(m) = \mathsf{T}$. □

Remark 4. Let F be a $\mathrm{PGA}_{\mathrm{ac}}$ instruction sequence and H be a service. Then $||F/_f H||$ is the process produced by F if F makes use of H. Instruction sequences that make use of services such as unbounded counters, unbounded stacks or Turing tapes are interesting because they may produce non-regular processes.

14 PGLD Programs and the Use of Boolean Registers

In this section, we show that all regular processes can also be produced by programs written in a program notation which is close to existing assembly languages, and even by programs in which no atomic action occurs more than once in an alternative choice instruction. The latter result requires programs that make use of Boolean registers.

A hierarchy of program notations rooted in PGA is introduced in [5]. One program notation that belongs to this hierarchy is PGLD, a very simple program notation which is close to existing assembly languages. It has absolute jump instructions and no explicit termination instruction.

In PGLD, like in PGA, it is assumed that there is a fixed but arbitrary finite set of *basic instructions* $\mathfrak{A}$. The primitive instructions of PGLD differ from the primitive instructions of PGA as follows: for each $l \in \mathbb{N}$, there is an *absolute jump instruction* $\#\#l$ instead of a forward jump instruction $\#l$. PGLD programs have the form $u_1; \ldots; u_k$, where $u_1, \ldots, u_k$ are primitive instructions of PGLD.

The effects of all instructions in common with PGA are as in PGA with one difference: if there is no next instruction to be executed, termination occurs. The effect of an absolute jump instruction $\#\#l$ is that execution proceeds with the l-th instruction of the program concerned. If $\#\#l$ is itself the l-th instruction, then deadlock occurs. If l equals 0 or l is greater than the length of the program, then termination occurs.

We define the meaning of PGLD programs by means of a function `pgld2pga` from the set of all PGLD programs to the set of all closed PGA terms. This function is defined by

$$\texttt{pgld2pga}(u_1 \,;\ldots;\, u_k) = (\phi_1(u_1) \,;\ldots;\, \phi_k(u_k) \,;\, !\,;\, !)^\omega \;,$$

where the auxiliary functions ϕ_j from the set of all primitive instructions of PGLD to the set of all primitive instructions of PGA are defined as follows $(1 \leq j \leq k)$:

$$\begin{aligned}
&\phi_j(\#\#l) = \#l - j && \text{if } j \leq l \leq k \;,\\
&\phi_j(\#\#l) = \#k + 2 - (j - l) && \text{if } 0 < l < j \;,\\
&\phi_j(\#\#l) = ! && \text{if } l = 0 \vee l > k \;,\\
&\phi_j(u) \quad = u && \text{if } u \text{ is not a jump instruction .}
\end{aligned}$$

PGLD is as expressive as PGA. Before we make this fully precise, we introduce a useful notation.

Let α is a valuation of variables in $\boldsymbol{\mathcal{I}}_{\mathrm{PGA}}$, and let α^* be the unique homomorphic extension of α to terms of PGA. Then $\alpha^*(t)$ is independent of α if t is

a closed term, i.e. $\alpha^*(t)$ is uniquely determined by $\mathcal{I}_{\mathrm{PGA}}$. Therefore, we write $t^{\mathcal{I}_{\mathrm{PGA}}}$ for $\alpha^*(t)$ if t is a closed term.

Proposition 6. *For each closed* PGA *term t, there exists a* PGLD *program p such that $|t^{\mathcal{I}_{\mathrm{PGA}}}| = |\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$.*

Proof. In [5], a number of functions (called embeddings in that paper) are defined, whose composition gives, for each closed PGA term t, a PGLD program p such that $|t^{\mathcal{I}_{\mathrm{PGA}}}| = |\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$. □

Let p be a PGLD program and P be a process. Then we say that p *produces* P if $|\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$ produces P.

Below, we will write $\mathrm{PGLD}_{\mathrm{ac}}$ for the version of PGLD in which the additional assumptions relating to $\mathfrak{A}$ mentioned in Section 11 are made. As a corollary of Theorem 2 and Proposition 6, we have that all regular processes over $\mathcal{AA}$ can be produced by $\mathrm{PGLD}_{\mathrm{ac}}$ programs.

Corollary 1. *Assume that CFAR is valid in $\mathcal{M}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists a* $\mathrm{PGLD}_{\mathrm{ac}}$ *program p such that p produces P.*

We switch to the use of Boolean registers now. First, we describe services that make up Boolean registers.

A Boolean register service accepts the following methods:

- a *set to true method* $\mathsf{set{:}T}$;
- a *set to false method* $\mathsf{set{:}F}$;
- a *get method* get.

We write $\mathcal{M}_{\mathsf{BR}}$ for the set $\{\mathsf{set{:}T}, \mathsf{set{:}F}, \mathsf{get}\}$. It is assumed that $\mathcal{M}_{\mathsf{BR}} \subseteq \mathcal{M}$.

The methods accepted by Boolean register services can be explained as follows:

- $\mathsf{set{:}T}$: the contents of the Boolean register becomes T and the reply is T;
- $\mathsf{set{:}F}$: the contents of the Boolean register becomes F and the reply is F;
- get : nothing changes and the reply is the contents of the Boolean register.

Let $s \in \mathbb{B} \cup \{\mathsf{B}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\mathbb{B} \cup \{\mathsf{B}\}, \mathit{eff}, \mathit{eff}, s)$, where the function eff is defined as follows ($b \in \mathbb{B}$):

$$\begin{aligned} &\mathit{eff}(\mathsf{set{:}T}, b) = \mathsf{T} \,, && \mathit{eff}(m, b) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{BR}} \,, \\ &\mathit{eff}(\mathsf{set{:}F}, b) = \mathsf{F} \,, && \mathit{eff}(m, \mathsf{B}) = \mathsf{B} \,. \\ &\mathit{eff}(\mathsf{get}, b) = b \,, \end{aligned}$$

Notice that the effect and yield functions of a Boolean register service are the same.

Let p be a PGLD program and P be a process. Then we say that p *produces P using Boolean registers* if $(\ldots(|\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}| \,/_{\mathsf{br}:1}\, BR_{\mathsf{F}}) \ldots /_{\mathsf{br}:k}\, BR_{\mathsf{F}})$ produces P for some $k \in \mathbb{N}^+$.

We have that $\mathrm{PGLD}_{\mathrm{ac}}$ programs in which no atomic action from $\mathcal{AA}$ occurs more than once in an alternative choice instruction can produce all regular processes over $\mathcal{AA}$ using Boolean registers.

Theorem 3. *Assume that CFAR is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^{\tau}+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists a $\mathrm{PGLD}_{\mathrm{ac}}$ program p in which each atomic action from $\mathcal{AA}$ occurs no more than once in an alternative choice instruction such that p produces P using Boolean registers.*

Proof. By the proof of Theorem 2 given in Section 12, it is sufficient to show that, for each thread T that is regular over $\mathcal{A}$, there exist a PGLD program p in which each basic action from $\mathcal{A}$ occurs no more than once and a $k \in \mathbb{N}^+$ such that $(\ldots(|\mathtt{pgld2pga}(p)^{\mathcal{Z}_{\mathrm{PGA}}}|\,/_{\mathsf{br}:1}\,BR_{\mathsf{F}})\ldots/_{\mathsf{br}:k}\,BR_{\mathsf{F}}) = T$.

Let T be a thread that is regular over $\mathcal{A}$. We may assume that T is produced by a PGLD program p' of the following form:

$$
\begin{array}{l}
+a_1\,;\#\#(3\cdot k_1+1)\,;\#\#(3\cdot k'_1+1)\,;\\
\quad\vdots\\
+a_n\,;\#\#(3\cdot k_n+1)\,;\#\#(3\cdot k'_n+1)\,;\\
\#\#0\,;\#\#0\,;\#\#0\,;\#\#(3\cdot n+4)\,,
\end{array}
$$

where, for each $i \in [1,n]$, $k_i, k'_i \in [0,n-1]$ (cf. the proof of Proposition 2 from [18]). It is easy to see that the PGLD program p that we are looking for can be obtained by transforming p': by making use of n Boolean registers, p can distinguish between different occurrences of the same basic instruction in p', and in that way simulate p'. □

15 Conclusions

Using the process algebra known as ACP, we have described two protocols to deal with the phenomenon that, on execution of an instruction sequence, a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. The more complex protocol is directed towards keeping the execution unit busy. In this way, we have brought the phenomenon better into the picture and have ascribed a sense to the term instruction stream which makes clear that an instruction stream is dynamic by nature, in contradistinction with an instruction sequence. We have also discussed some conceivable adaptations of the more complex protocol.

The description of the protocol starts from the behaviours produced by instruction sequences under execution. By that we abstract from the instruction sequences which produce those behaviours. How instruction streams can be generated efficiently from instruction sequences is a matter that obviously requires investigations at a less abstract level. The investigations in question are an option for future work.

We believe that the more complex protocol described in this paper provides a setting in which basic techniques aimed at increasing processor performance,

such as pre-fetching and branch-prediction, can be studied at a more abstract level than usual (cf. [14]). In particular, we think that the protocol can serve as a starting-point for the development of a model with which trade-offs encountered in the design of processor architectures can be clarified. We consider investigations into this matter an interesting option for future work.

Because process algebras such as ACP, CCS and CSP are considered relevant to computer science, there must be programmed systems whose behaviours are taken for processes as considered in such a process algebra. In that light, we have investigated the connections between programs and the processes that they produce, starting from the perception of a program as an instruction sequence. We have shown that, by apposite choice of basic instructions, all regular processes can be produced by means of instruction sequences as considered in PGA.

We have also made precise what processes are produced by instruction sequences under execution that make use of services. The reason for this is that instruction sequences under execution are regular threads and regular threads that make use of services such as unbounded counters, unbounded stacks or Turing tapes may produce non-regular processes. An option for future work is to characterize the classes of processes that can be produced by single-pass instruction sequences that make use of such services.

References

1. Baeten, J.C.M., Bergstra, J.A.: Process algebra with signals and conditions. In: Broy, M. (ed.) Programming and Mathematical Methods. NATO ASI Series, vol. F88, pp. 273–323. Springer-Verlag (1992)
2. Baeten, J.C.M., Weijland, W.P.: Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
3. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Proceedings 30th ICALP. Lecture Notes in Computer Science, vol. 2719, pp. 1–21. Springer-Verlag (2003)
4. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1–3), 109–137 (1984)
5. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming 51(2), 125–156 (2002)
6. Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Formal Aspects of Computing 19(4), 445–474 (2007)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequences for the production of processes. `arXiv:0811.0436v2 [cs.PL]` (November 2008)
8. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic 6(4), 553–563 (2008)
9. Bergstra, J.A., Middelburg, C.A.: A protocol for instruction stream processing. `arXiv:0905.2257v1 [cs.PL]` (May 2009)
10. Bergstra, J.A., Middelburg, C.A.: Transmission protocols for instruction streams. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. Lecture Notes in Computer Science, vol. 5684, pp. 127–139. Springer-Verlag (2009)
11. Bergstra, J.A., Middelburg, C.A.: On the expressiveness of single-pass instruction sequences. Theory of Computing Systems (2010), DOI: 10.1007/s00224-010-9301-8

12. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM 31(3), 560–599 (1984)
13. Fokkink, W.J.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin (2000)
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, third edn. (2003)
15. Hennessy, M., Milner, R.: Algebraic laws for non-determinism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
17. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
18. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: Beckmann, A., et al. (eds.) CiE 2006. Lecture Notes in Computer Science, vol. 3988, pp. 445–458. Springer-Verlag (2006)